



University of
Pittsburgh

Algorithms and Data Structures 2

CS 1501



Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
 - Lab 6: tonight 10/31 @ 11:59 pm
 - **Assignment 2: this Friday 11/4 @ 11:59 pm**
 - Homework 8: this Friday @ 11:59 pm (not posted yet)
 - Lab 7: next Monday 11/7 @ 11:59 pm
- Live Support Session for Assignment 2
 - Recording and slides on the assignment Canvas page
- Weekly Live QA Session on Piazza
 - Friday 4:30-5:30 pm

Previous lecture

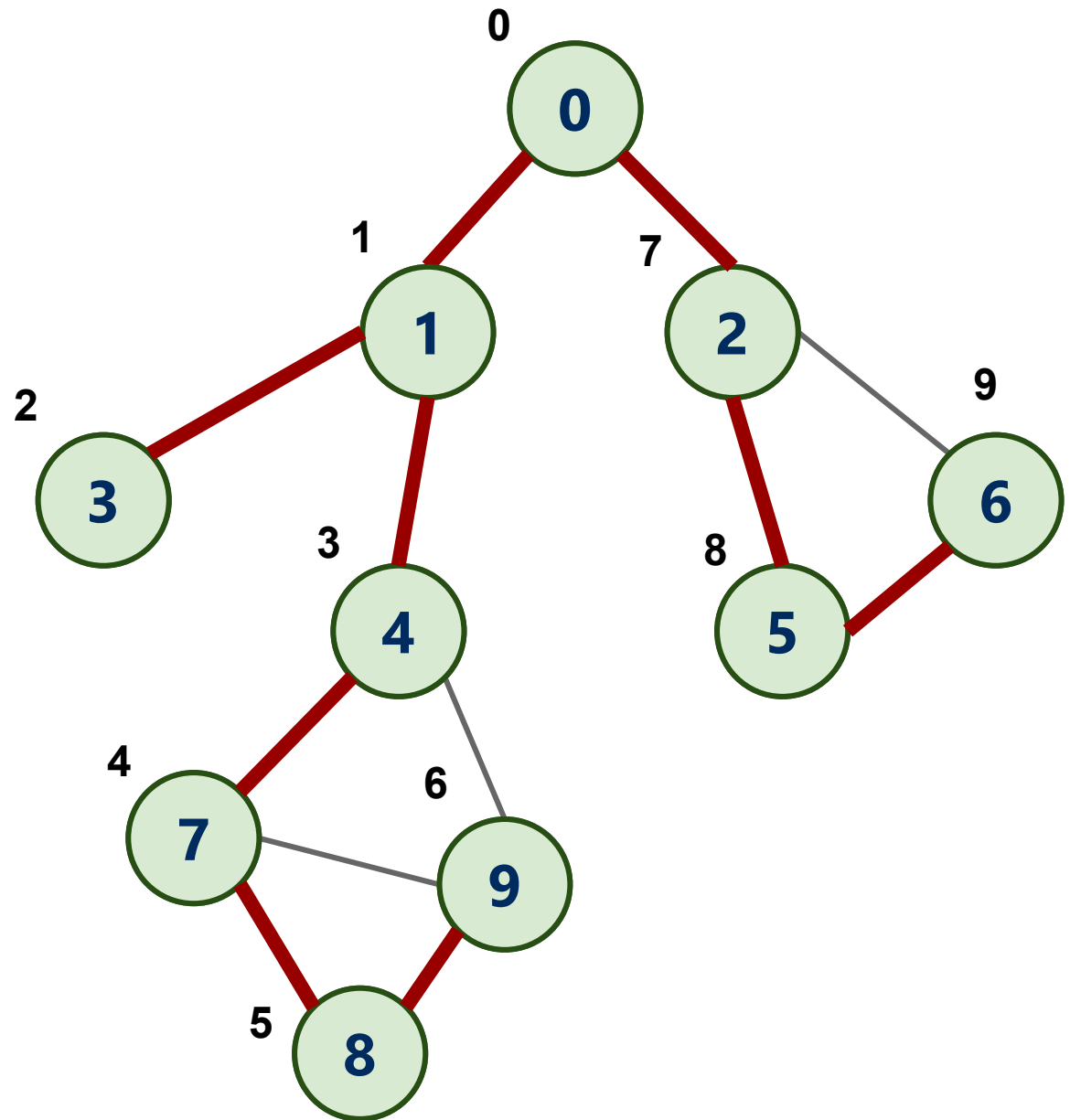
- ADT Graph
 - traversals
 - DFS
 - finding articulation points of a graph

This Lecture

- ADT Graph
 - finding articulation points of a graph
 - Graph compression
 - Graphs with weighted edges
 - Minimum Spanning Tree (MST) problem
 - Prim's MST algorithm

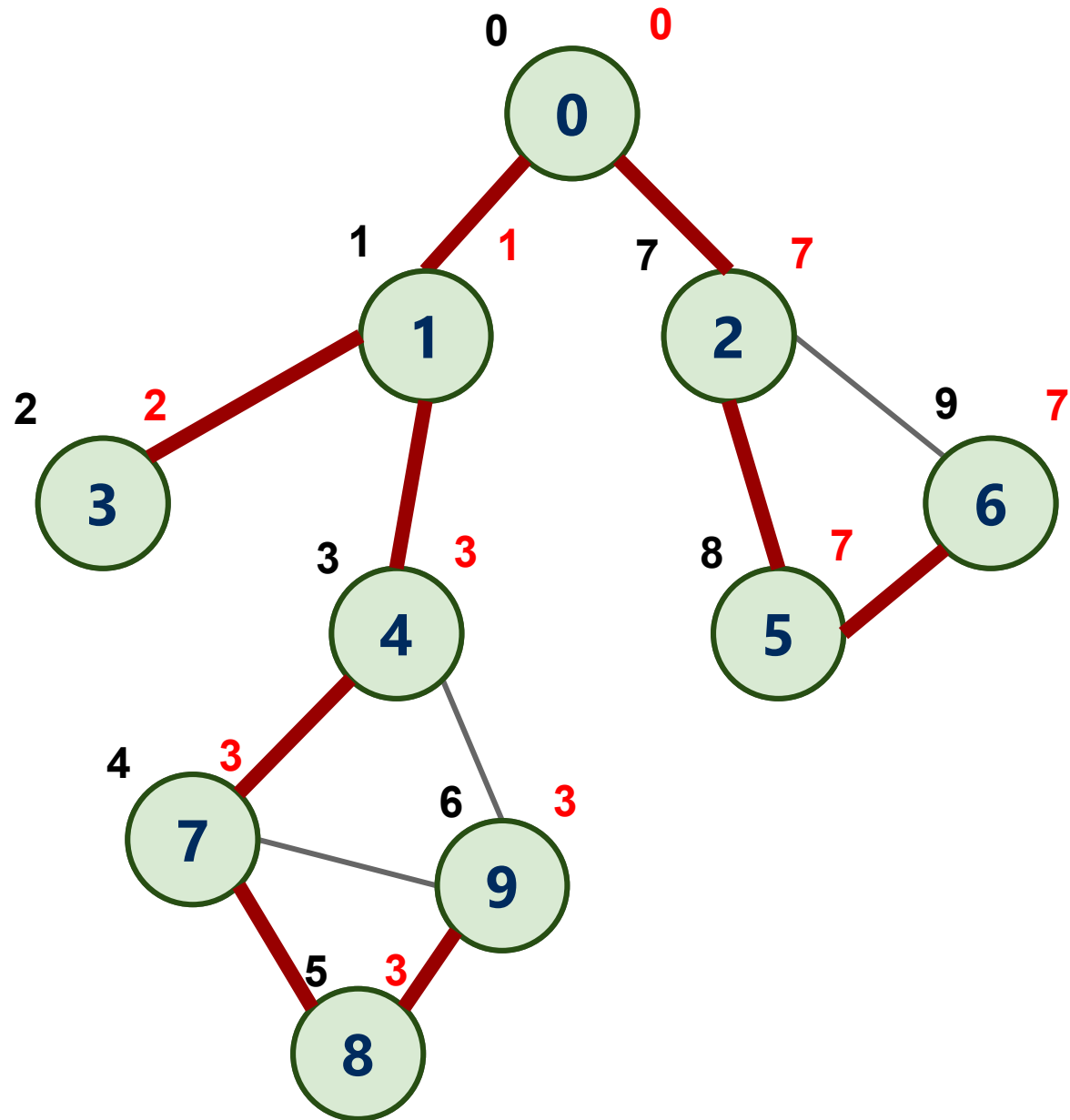
num(v)

- A pre-order DFS traversal visits the vertices in some order
 - let's number the vertices with their traversal order
 - num(v)



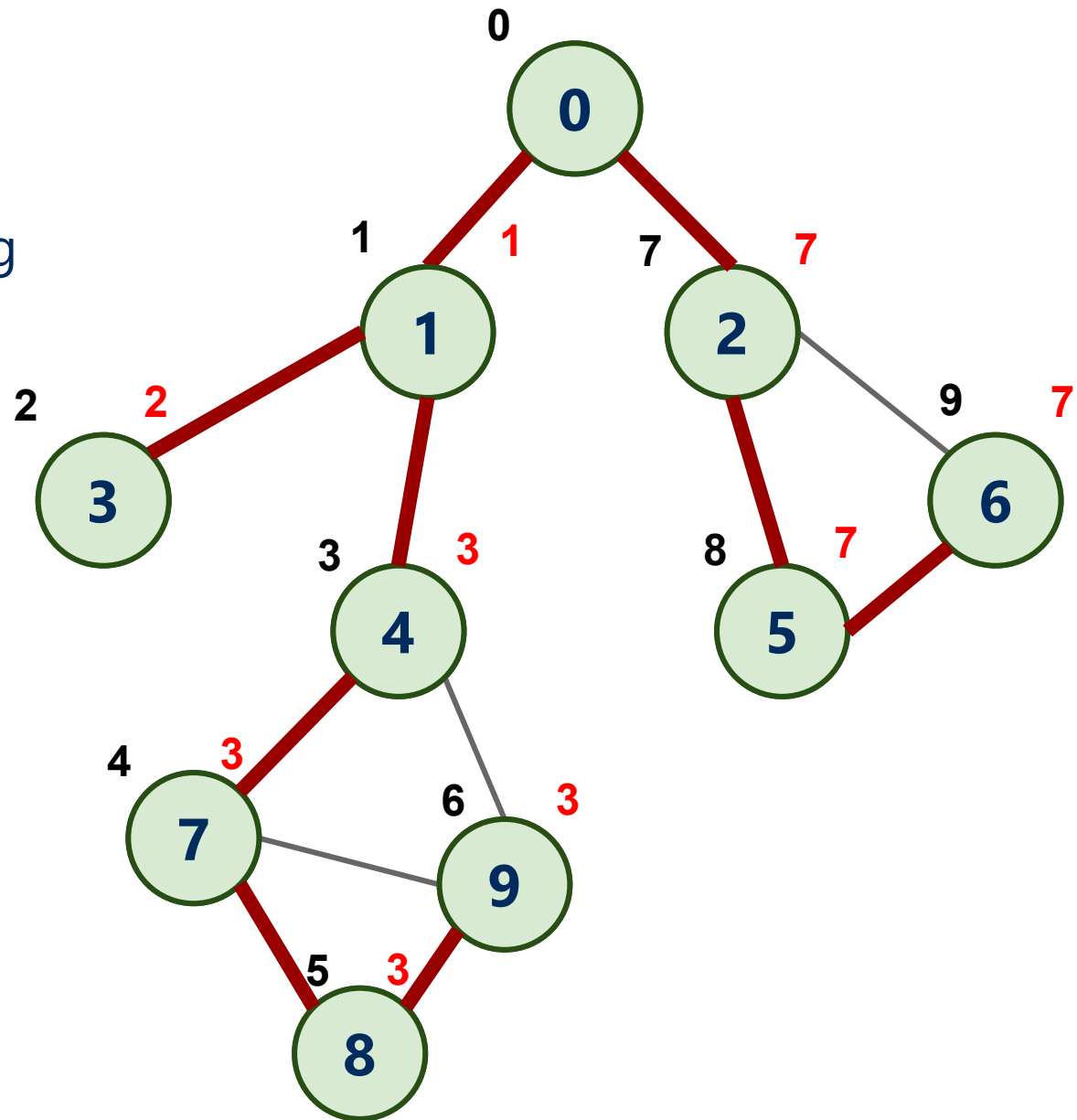
low(v)

- Edges not included in the spanning tree built by DFS are called **back edges**
 - e.g., (4, 9) and (2, 6)
- For each vertex v , find the lowest numbered vertex reachable from v through vertices that have not been seen yet
 - $\text{low}(v)$



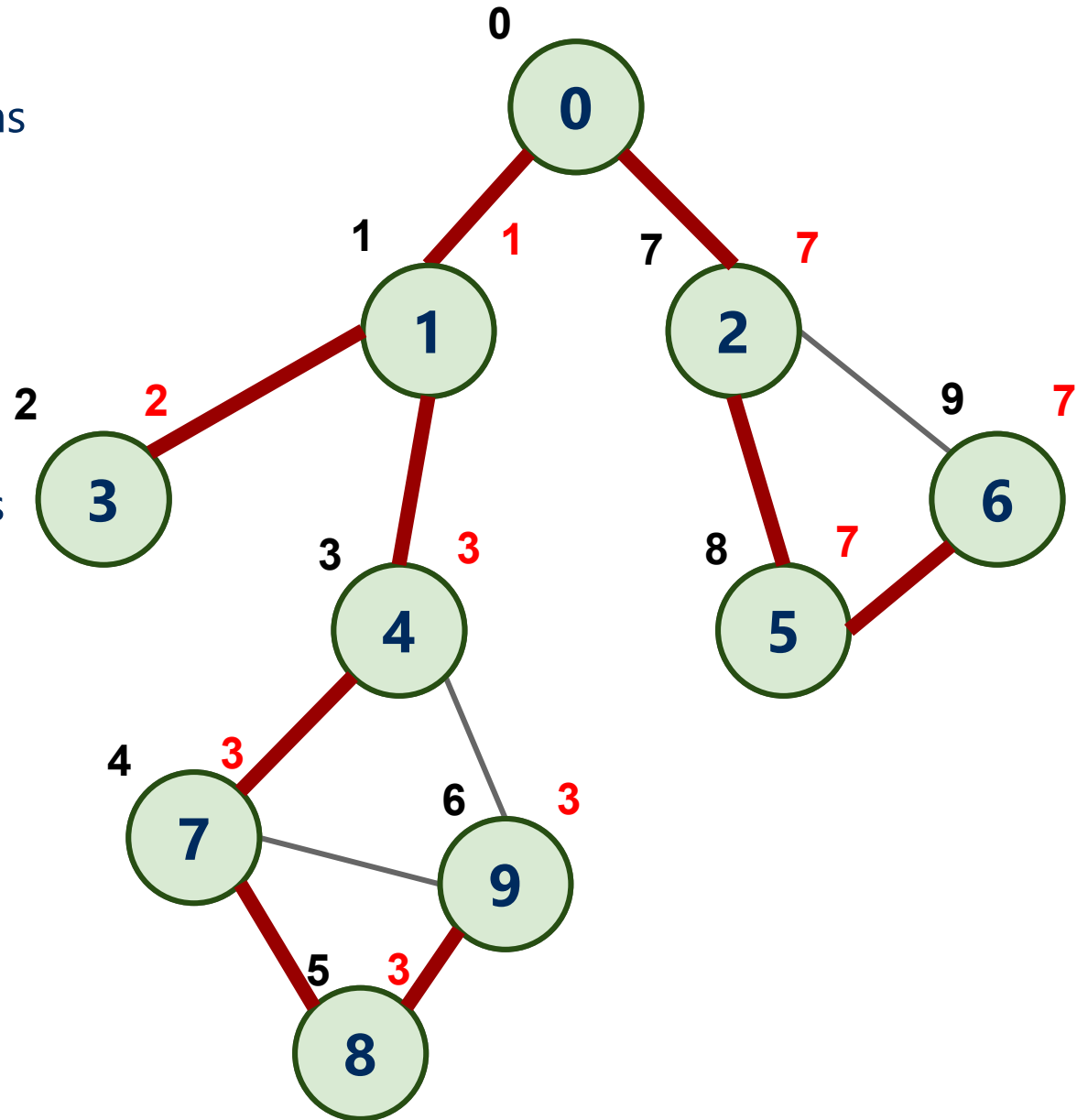
low(v)

- How did you find low(v)?
- Move down the tree looking for a back edge that goes backwards “the furthest”
- Used
 - **using 0 or more tree edges going down**
 - **then at most one back edge**



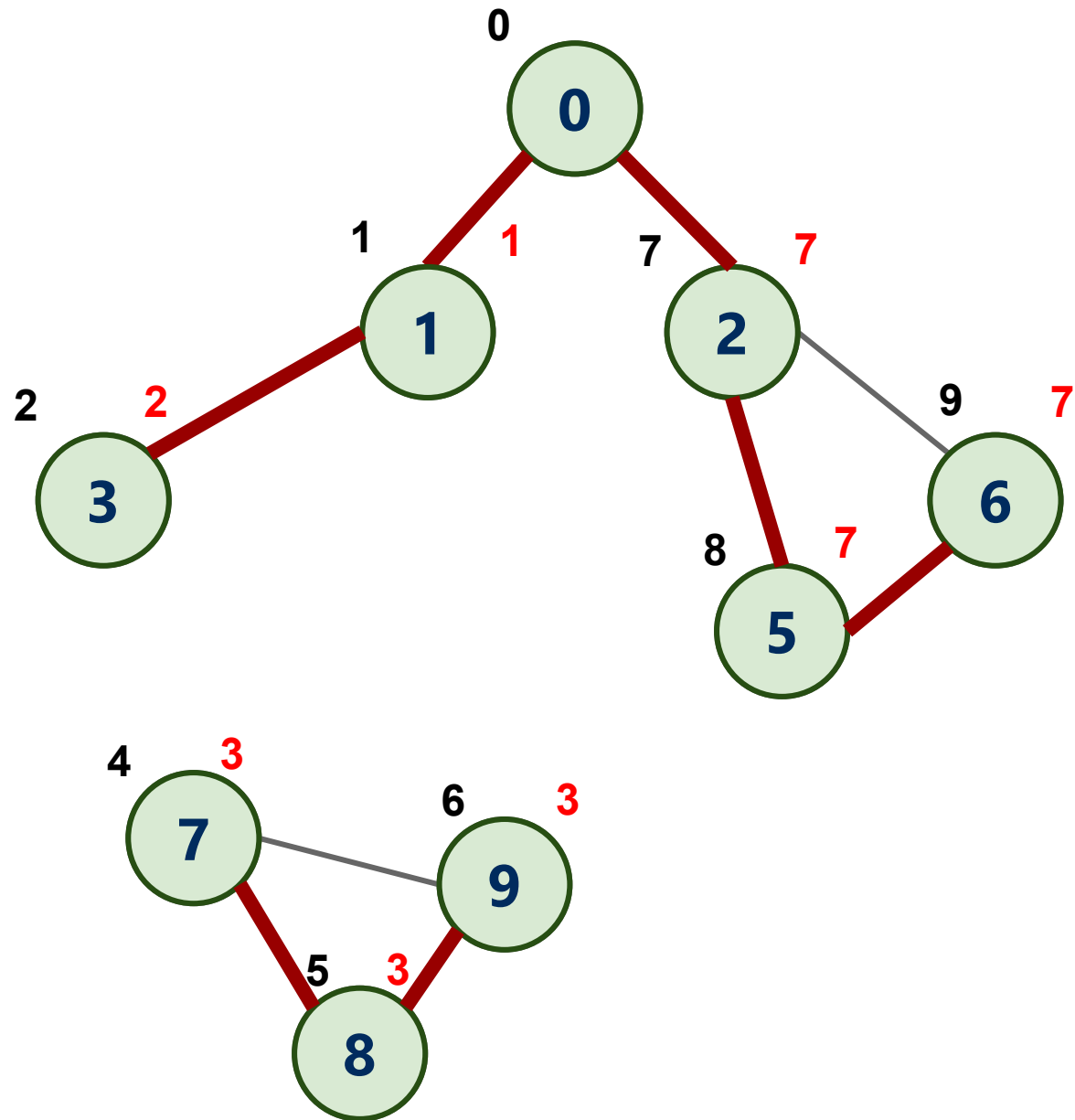
Finding articulation points of a graph

- What does it mean if a vertex has a child such that
 - $\text{low}(\text{child}) \geq \text{num}(\text{parent})$?
- e.g., 4 and 7
- child has no other way except through parent to reach vertices with lower num values than parent
- e.g., 7 cannot reach 0, 1, and 3 except through 4
- So, the parent is an articulation point!
 - e.g., if 4 is removed, the graph becomes disconnected



Finding articulation points of a graph

- if 4 is removed, the graph becomes disconnected
- Each non-root vertex v that has a child w such that $\text{low}(w) \geq \text{num}(v)$ is an articulation point
- What about the root vertex?
- The root has the smallest num value
 - can't go "further" than root
 - need a different condition for root



low(v)

- $\text{low}(v)$ = the num value of the lowest-numbered vertex reachable from v using 0 or more spanning tree edges and then **at most one** back edge
 - Min of:
 - $\text{num}(v)$ (the vertex is reachable from itself)
 - $\text{num}(w)$ for all back edges (v, w)
 - $\text{low}(w)$ of all children of v (the lowest-numbered vertex reachable through a child)

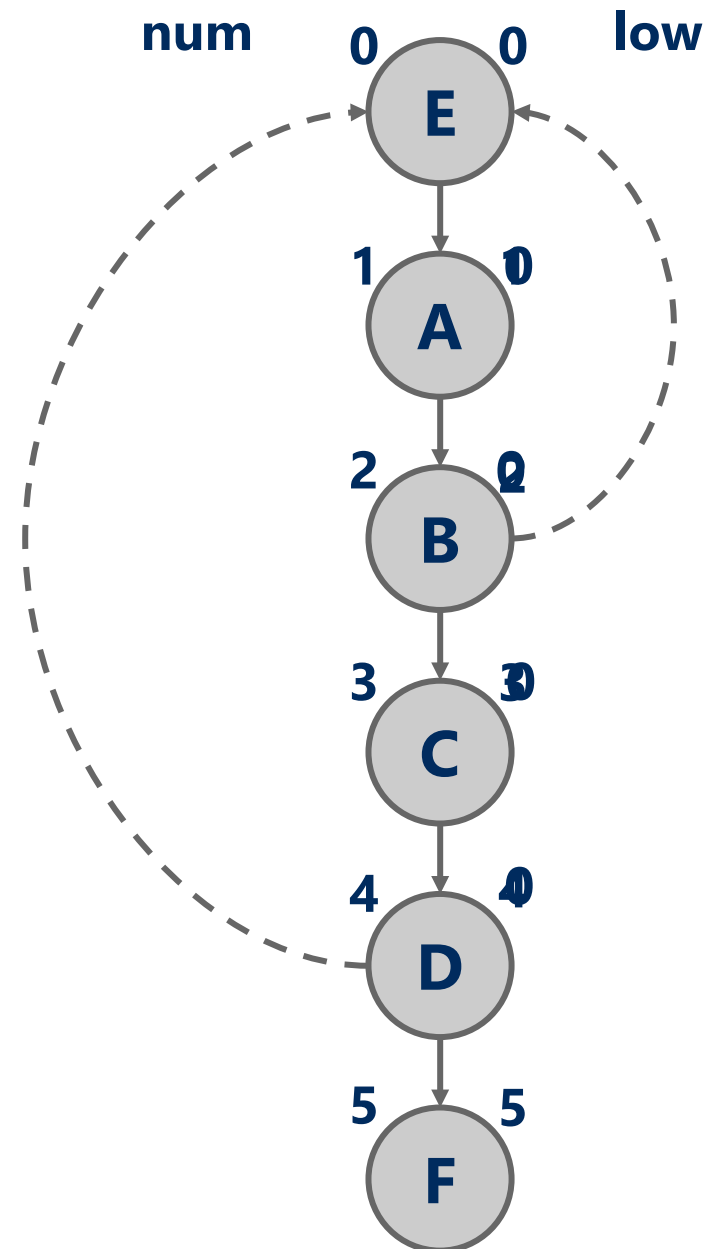
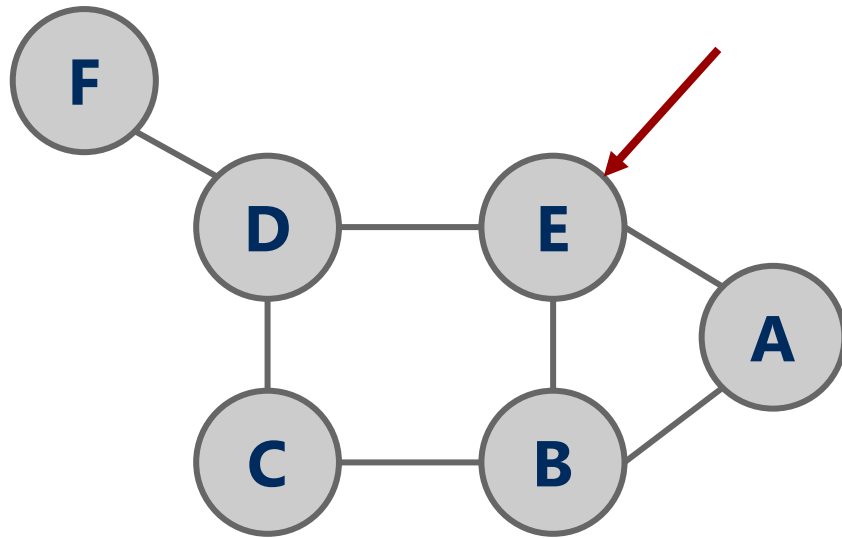
Finding articulation points of a graph: The Algorithm

- As DFS visits each vertex v
 - Label v with the two numbers:
 - $\text{num}(v)$
 - $\text{low}(v)$: initial value is $\text{num}(v)$
 - For each neighbor w
 - if already seen \rightarrow we have a back edge
 - update $\text{low}(v)$ to $\text{num}(w)$ if $\text{num}(w)$ is less
 - if not seen \rightarrow we have a child
 - call DFS on the child
 - **after the call returns,**
 - update $\text{low}(v)$ to $\text{low}(w)$ if $\text{low}(w)$ is less

when to compute $\text{num}(v)$ and $\text{low}(v)$

- $\text{num}(v)$ is computed as we move down the tree
 - pre-order DFS
- $\text{low}(v)$ is updated as we move down and up the tree
- Recursive DFS is convenient to compute both
 - why?

Finding articulation points example



Using DFS to find the articulation points of a connected undirected graph

```
int num = 0
```

```
DFS(vertex v) {
```

```
    num[v] = num++
```

```
    low[v] = num[v] //initially
```

```
    seen[v] = true //mark v as seen
```

```
    for each neighbor w
```

```
        if(w unseen){
```

```
            parent[w] = v
```

```
            DFS(w) //after the call returns low[w] is computed, why?
```

```
            low[v] = min(low[v], low[w])
```

```
        } else { //seen neighbor
```

```
            if(w != parent[v]) //and not the parent, so back edge
```

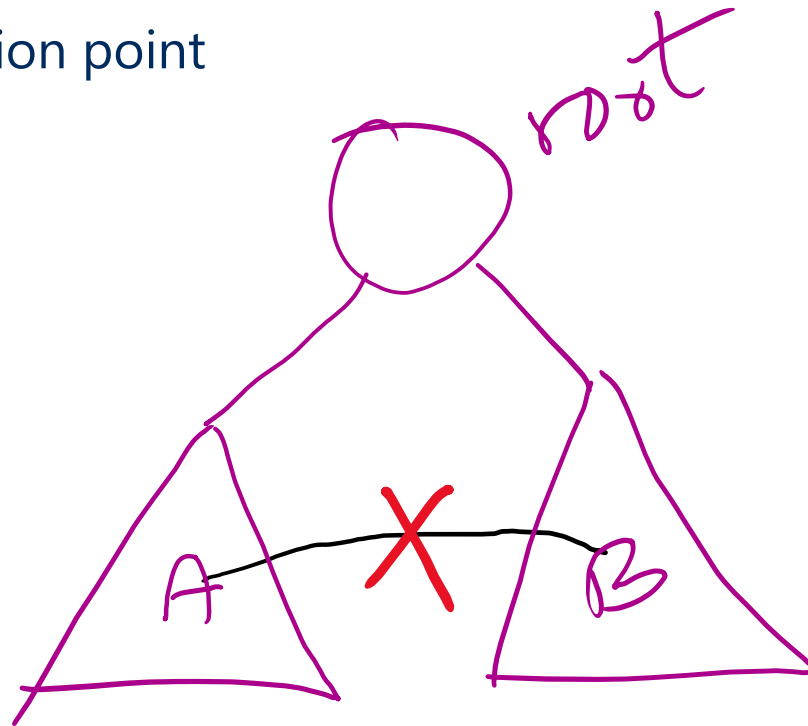
```
                low[v] = min(low[v], num[w])
```

```
        }
```

```
}
```

What about the root of the spanning tree?

- What if we start DFS at an articulation point?
 - The starting vertex becomes the root of the spanning tree
 - If the root of the spanning tree has more than one child, the root is an articulation point

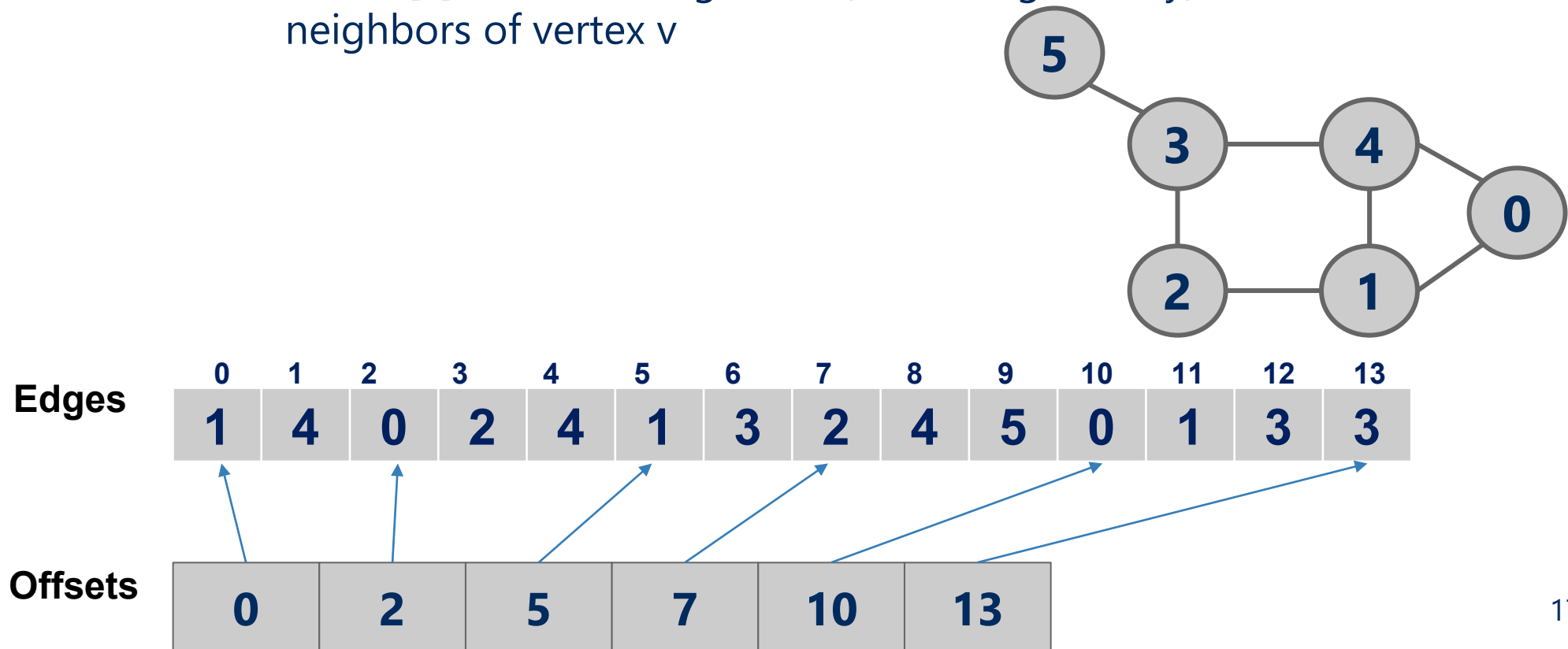


Graph Compression

- Real-life graphs are huge
 - 100's if not 1000's of GBs
 - Facebook graph, Google graph, maps, ...
- Let's see one (partial) idea for reducing the size of large graphs

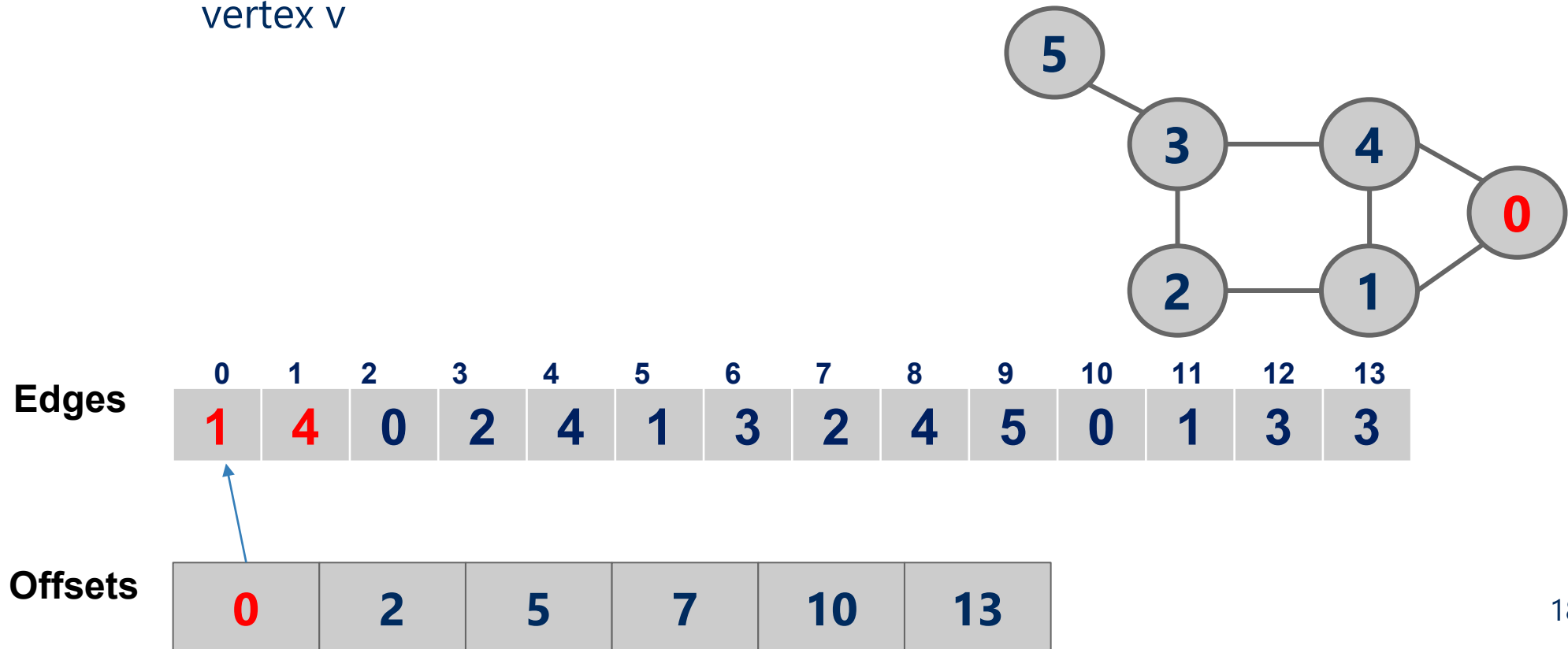
Graph Compression

- **Step 1:** Construct a Compressed Sparse Row (CSR) representation of the graph
- CSR
 - Edges array concatenates *sorted* neighbor lists of all vertices
 - Offsets array:
 - $\text{offsets}[v]$ is the starting index (in the Edges array) for the neighbors of vertex v



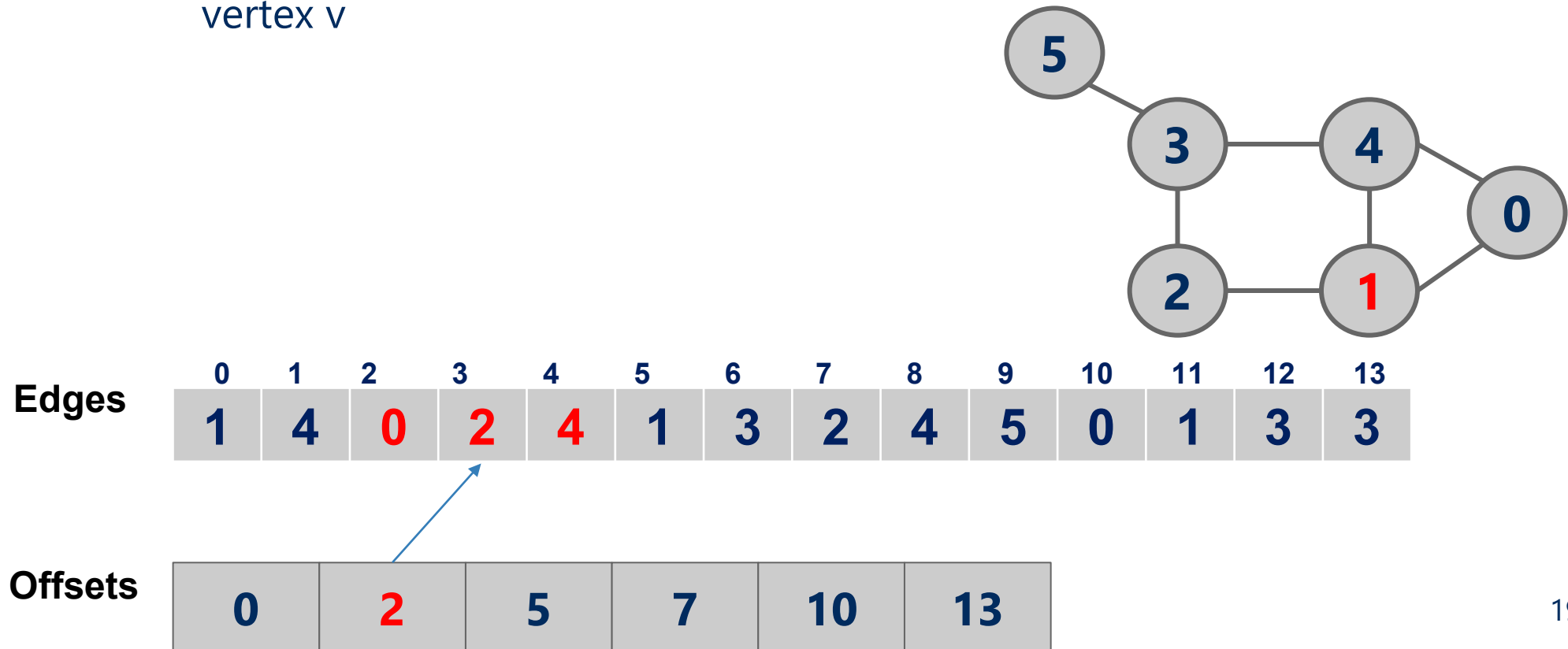
Graph Compression

- Let's start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
 - $\text{offsets}[v]$ is the starting index (in the Edges array) for the neighbors of vertex v



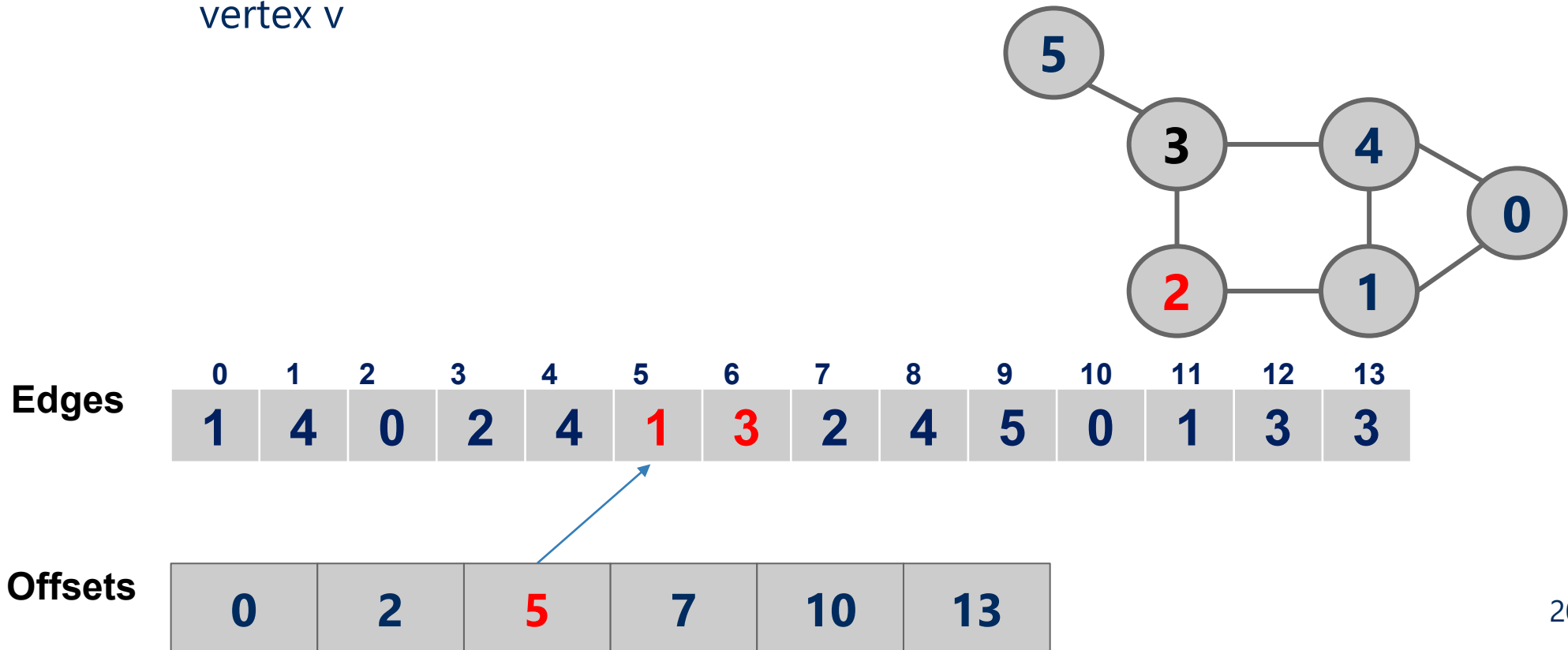
Graph Compression

- Let's start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
 - $\text{offsets}[v]$ is the starting index (in the Edges array) for the neighbors of vertex v



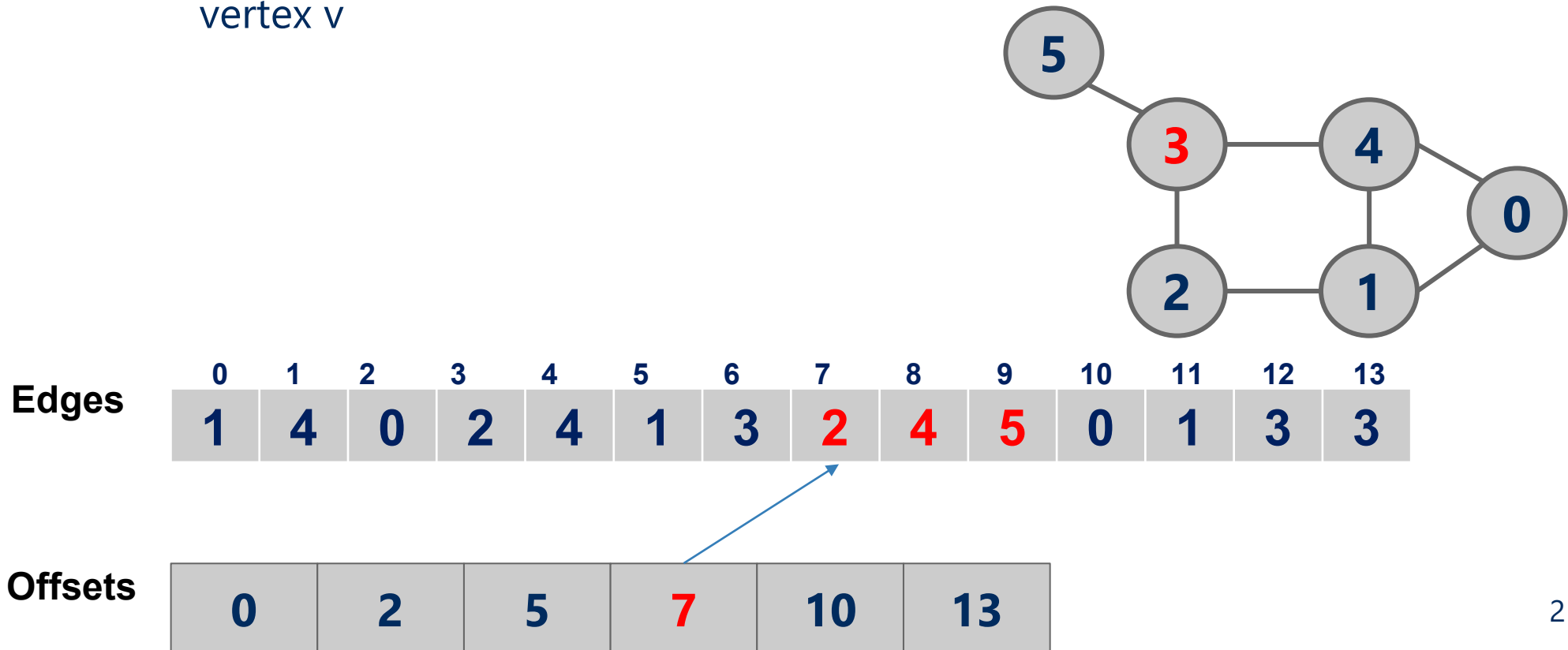
Graph Compression

- Let's start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
 - $\text{offsets}[v]$ is the starting index (in the Edges array) for the neighbors of vertex v



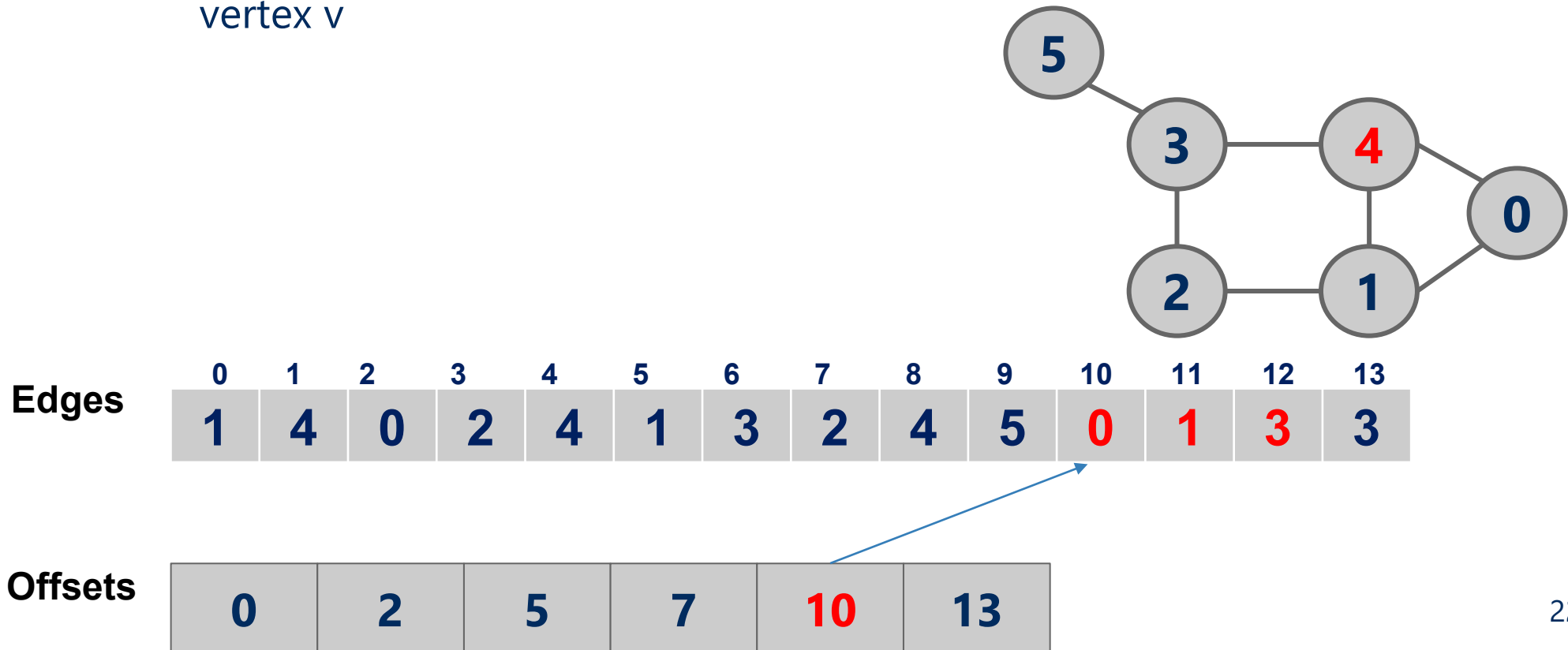
Graph Compression

- Let's start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
 - $\text{offsets}[v]$ is the starting index (in the Edges array) for the neighbors of vertex v



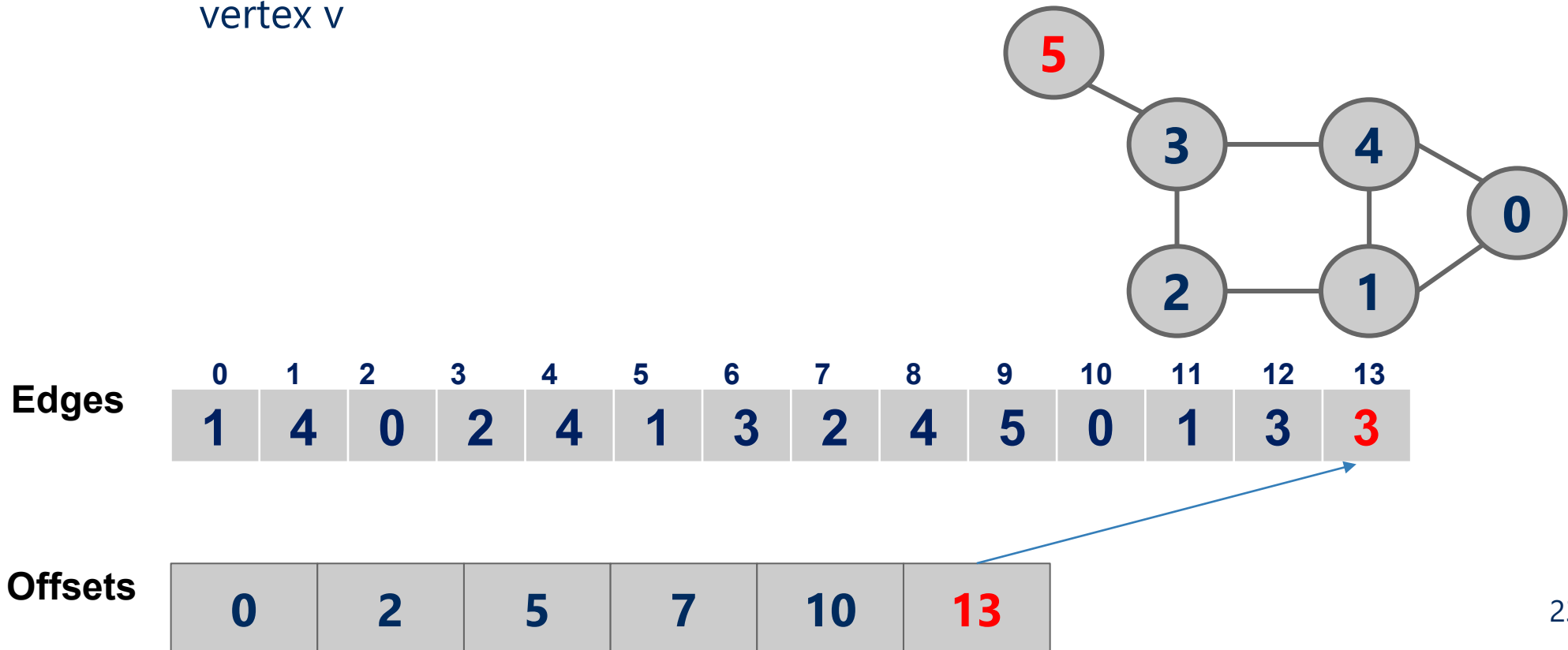
Graph Compression

- Let's start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
 - $\text{offsets}[v]$ is the starting index (in the Edges array) for the neighbors of vertex v



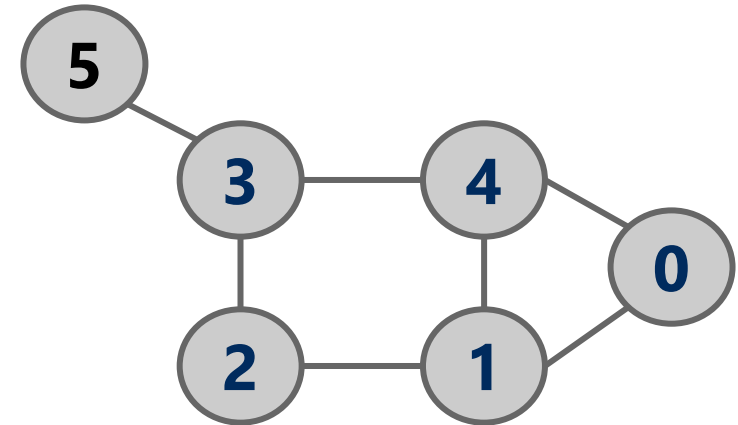
Graph Compression

- Let's start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
 - $\text{offsets}[v]$ is the starting index (in the Edges array) for the neighbors of vertex v



Graph Compression

- Can we compute the degree of a vertex using the offsets array?
 - Running time?
- What is the required space of this representation?
 - $\Theta(m + n)$
 - Assume 4 bytes per vertex and per edge
 - Total size: $4*v + 8*e$ bytes



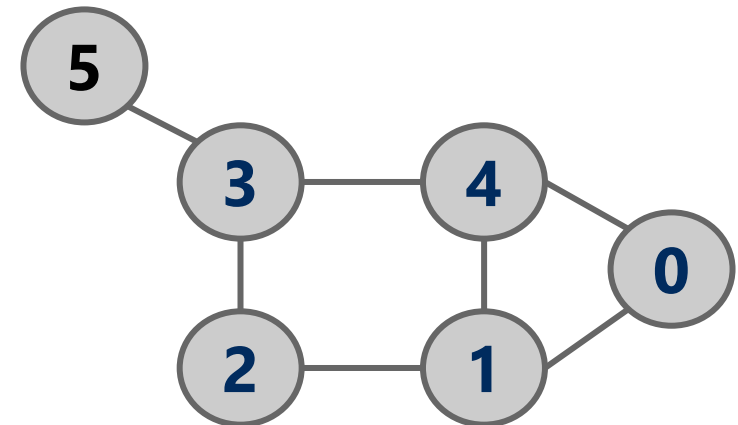
Edges	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets	0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**

- For each vertex v , with a neighbor list v_1, v_2, v_3, \dots
- Store the differences between each two consecutive numbers
 - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \dots$



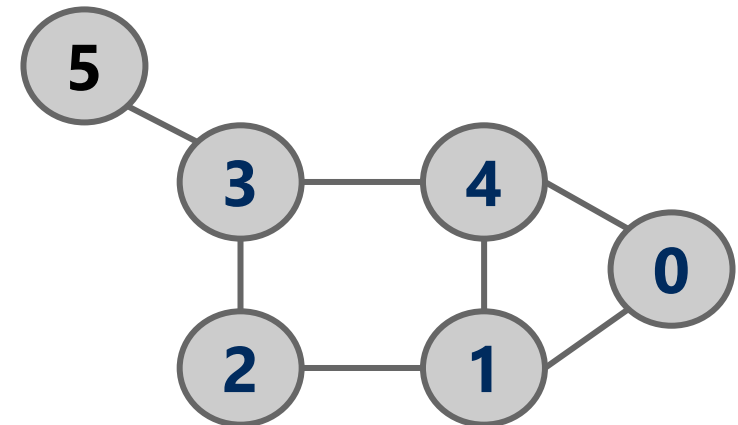
Edges	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets	0	2	5	7	10	13
	0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**

- For each vertex v , with a neighbor list v_1, v_2, v_3, \dots
- Store the differences between each two consecutive numbers
 - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \dots$

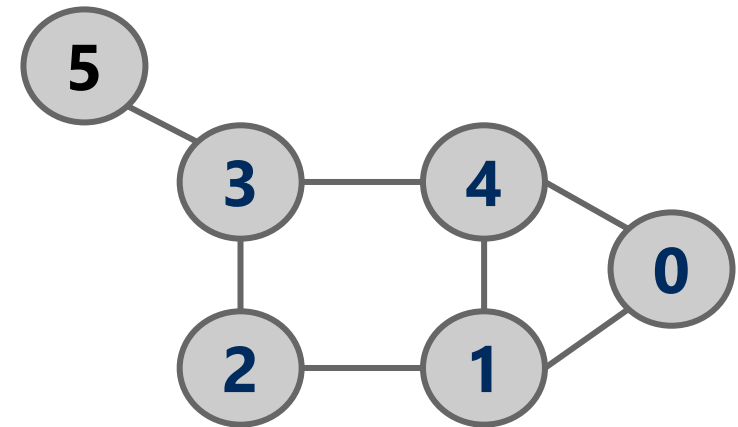


Edges	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	4	0	2	4	1	3	2	4	5	0	1	3	3
	1-0	4-1												
Offsets	0	2	5	7	10	13								

Graph Compression

- **Step 2: Difference coding**

- For each vertex v , with a neighbor list v_1, v_2, v_3, \dots
- Store the differences between each two consecutive numbers
 - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \dots$

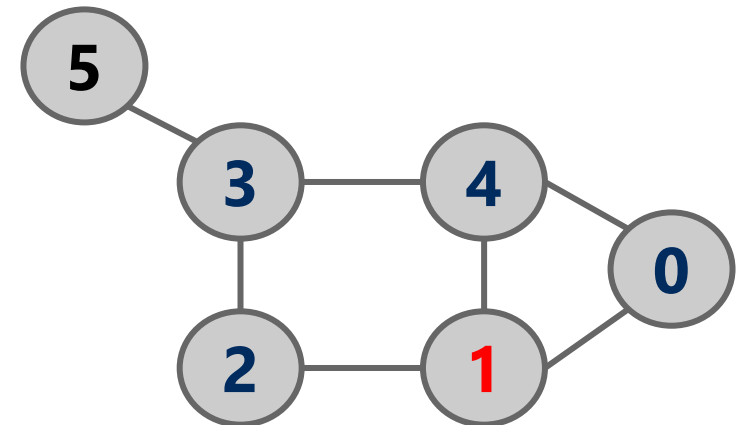


Edges	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	4	0	2	4	1	3	2	4	5	0	1	3	3
	1	3												
Offsets	0	2	5	7	10	13								

Graph Compression

- **Step 2: Difference coding**

- For each vertex v , with a neighbor list v_1, v_2, v_3, \dots
- Store the differences between each two consecutive numbers
 - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \dots$

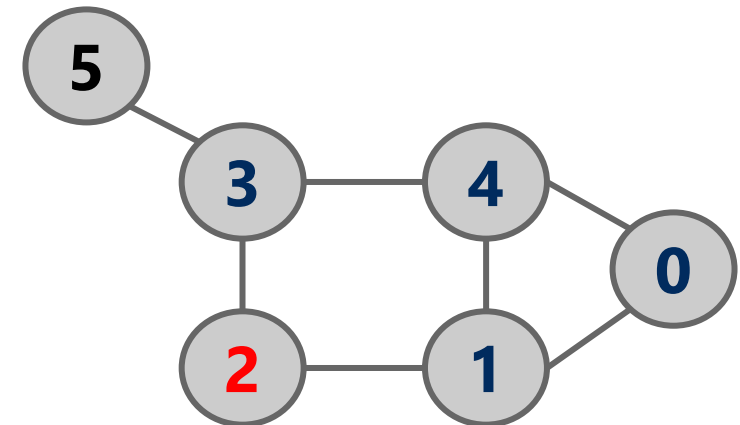


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Edges	1	4	0	2	4	1	3	2	4	5	0	1	3	3
	1	3	-1	2	2									
Offsets	0	2	5	7	10	13								

Graph Compression

- **Step 2: Difference coding**

- For each vertex v , with a neighbor list v_1, v_2, v_3, \dots
- Store the differences between each two consecutive numbers
 - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \dots$

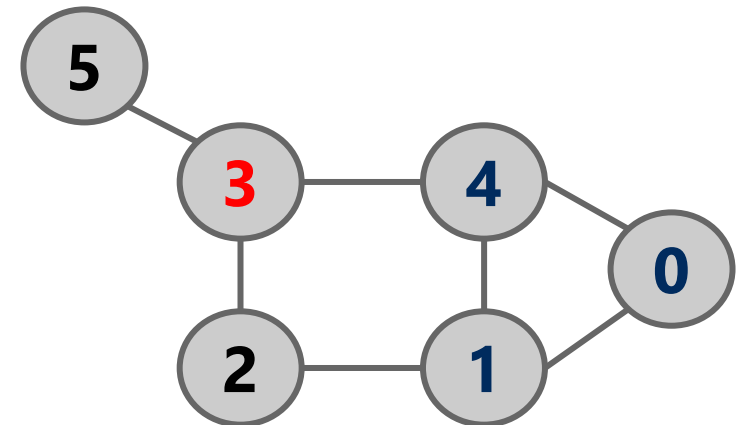


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Edges	1	4	0	2	4	1	3	2	4	5	0	1	3	3
	1	3	-1	2	2	-1	2							
Offsets	0	2	5	7	10	13								

Graph Compression

- **Step 2: Difference coding**

- For each vertex v , with a neighbor list v_1, v_2, v_3, \dots
- Store the differences between each two consecutive numbers
 - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \dots$

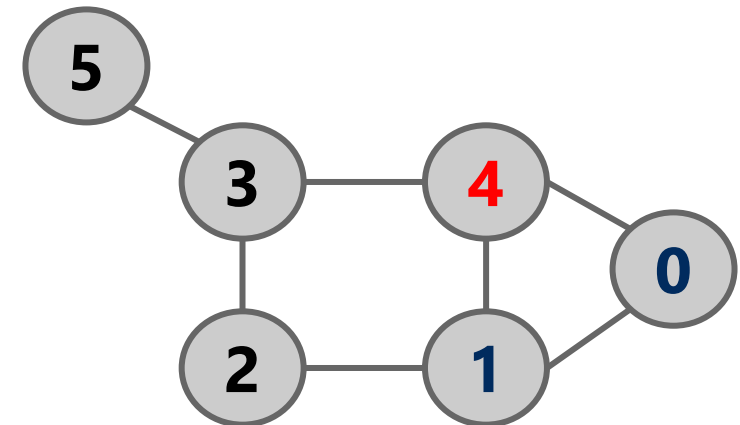


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Edges	1	4	0	2	4	1	3	2	4	5	0	1	3	3
	1	3	-1	2	2	-1	2	-1	2	1				
Offsets	0	2	5	7	10	13								

Graph Compression

- **Step 2: Difference coding**

- For each vertex v , with a neighbor list v_1, v_2, v_3, \dots
- Store the differences between each two consecutive numbers
 - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \dots$

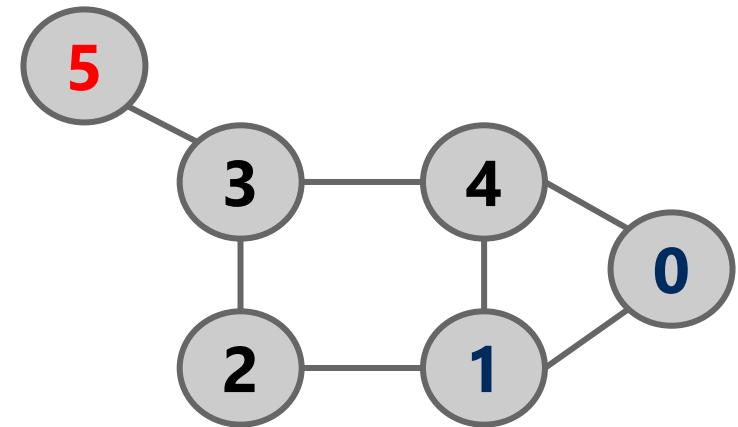


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Edges	1	4	0	2	4	1	3	2	4	5	0	1	3	3
	1	3	-1	2	2	-1	2	-1	2	1	-4	1	2	
Offsets	0	2	5	7	10	13								

Graph Compression

- **Step 2: Difference coding**

- For each vertex v , with a neighbor list v_1, v_2, v_3, \dots
- Store the differences between each two consecutive numbers
 - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \dots$

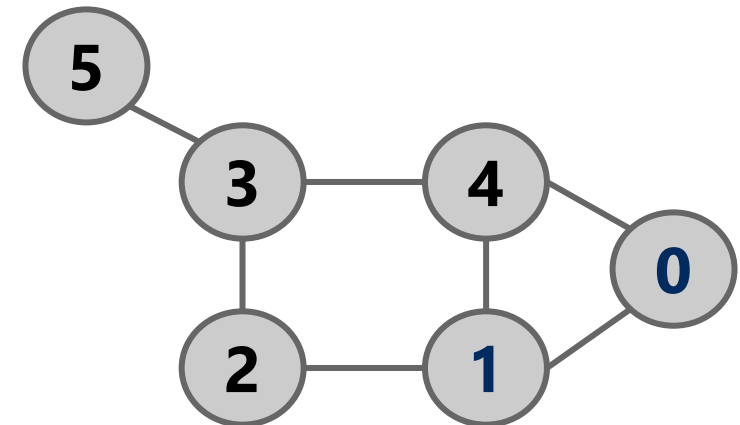


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Edges	1	4	0	2	4	1	3	2	4	5	0	1	3	3
	1	3	-1	2	2	-1	2	-1	2	1	-4	1	2	-2
Offsets	0	2	5	7	10	13								

Graph Compression

- **Step 2: Difference coding**

- For each vertex v , with a neighbor list v_1, v_2, v_3, \dots
- Store the differences between each two consecutive numbers
 - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \dots$

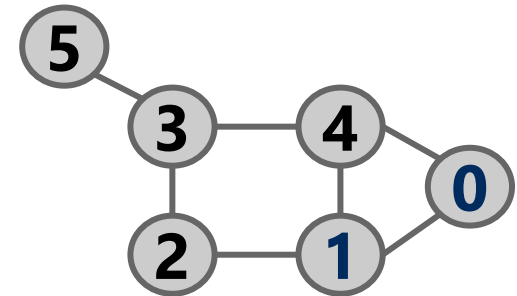


Edges	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	3	-1	2	2	-1	2	-1	2	1	-4	1	2	-2

Offsets	0	2	5	7	10	13
---------	---	---	---	---	----	----

Graph Compression

- Step 3: Use Gamma code to compress the differences



Edges	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	3	-1	2	2	-1	2	-1	2	1	-4	1	2	-2

Offsets	0	2	5	7	10	13
---------	---	---	---	---	----	----

Gamma Code

- Gamma Code is used to compress data in which small values are much more frequent than large values
- To encode an integer x ,
 - find T , the largest power of 2 $< x$
 - Encode T as $(\log T)$ zeros followed by 1
 - Append the remaining $(\log T)$ binary digits of x
- Example: To encode 17: 10001
 - $T = 16 = 2^4$
 - Gamma code: 0000 1 0001
- $2 \lfloor \log x \rfloor + 1$
 - much smaller than 32 bits if x is small

Graph Compression

- **Goal:** make the differences between vertex labels in each neighbor list small
 - So that their Gamma codes are much less than 32 bits
- For Web Graphs
 - Each vertex is a web page
 - Sort the pages based on their reverse URL (e.g., www.cs.pitt.edu)
 - Most links are local (within the same domain)
 - neighbors will be close to each other in the sorted list
 - Goal achieved
- Other graphs can be relabeled to achieve that goal
 - <https://www.cs.cmu.edu/~guyb/papers/BBK03.pdf>

Neighborhood connectivity Problem

- We want to keep a set of neighborhoods connected with the minimum cost possible
- **Input:** A set of neighborhoods and a file with the following format:
 - neighborhood i, neighborhood j, cost of connecting the two neighborhoods
 - ...
- **Output:** A set of neighborhood pairs to be connected and a total cost such that
 - We can go from any neighborhood to any other (**connected**)
 - The total cost should be minimum (i.e., as small as it can be) (**minimal cost**)

Think Data Structures First!

- How can we structure the input in computer memory?
- Can we use Graphs?
- What about the costs? How can we model that?

We said spatial layouts of graphs were irrelevant

- We define graphs as sets of vertices and edges
- However, we'll certainly want to be able to reason about bandwidth, distance, capacity, etc. of the real world things our graph represents
 - Whether a link is 1 gigabit or 10 megabit will drastically affect our analysis of traffic flowing through a network
 - Having a road between two cities that is a 1 lane country road is very different from having a 4 lane highway
 - If two airports are 2000 miles apart, the number of flights going in and out between them will be drastically different from airports 200 miles apart

We can represent such information with edge weights

- How do we store edge weights?
 - Adjacency matrix?
 - Adjacency list?
 - Do we need a whole new graph representation?
- How do weights affect finding spanning trees/shortest paths?
 - The weighted variants of these problems are called finding the *minimum spanning tree* and the *weighted shortest path*

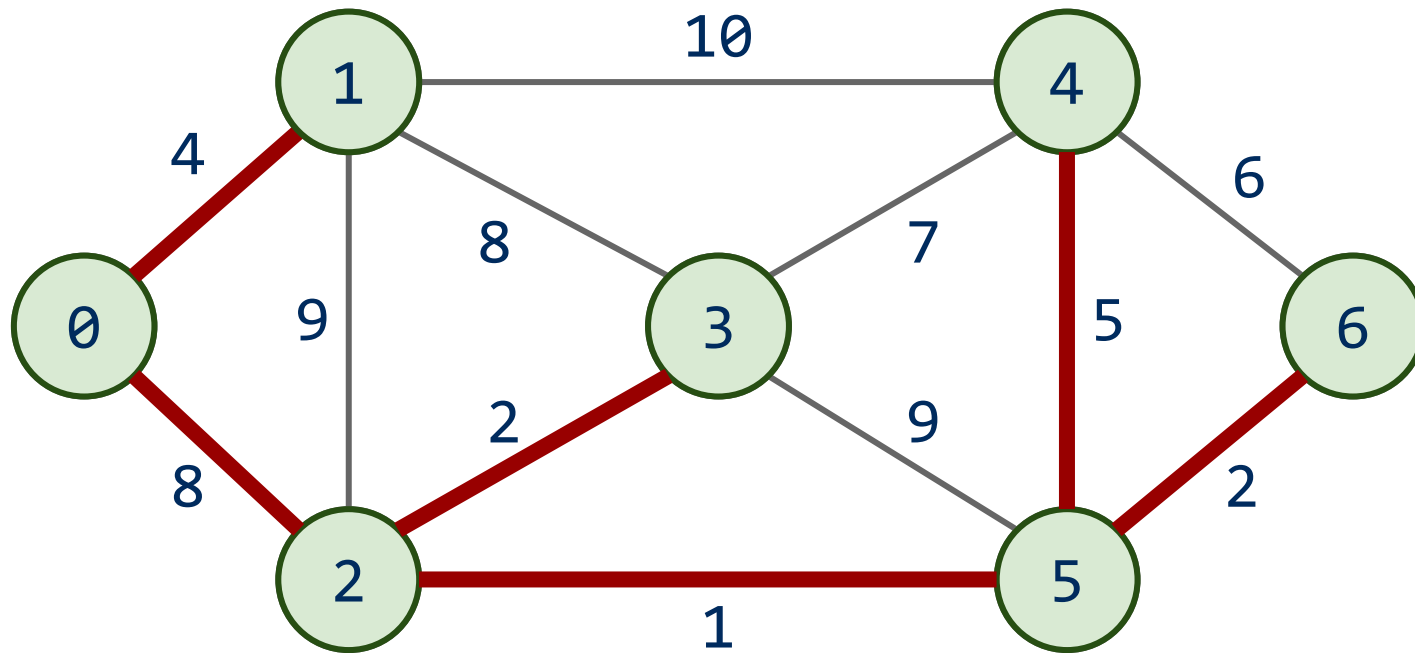
Minimum spanning trees (MST)

- Graphs can potentially have multiple spanning trees
- MST is the spanning tree that has the minimum sum of the weights of its edges

Prim's algorithm

- Initialize T to contain the starting vertex
 - T will eventually become the MST
- While there are vertices not in T :
 - Find minimum edge-weight edge that connects a vertex in T to a vertex not yet in T
 - Add the edge with its vertex to T

Prim's algorithm



Runtime of Prim's

- At each step, check all possible edges
- For a complete graph:
 - First iteration:
 - $v - 1$ possible edges
 - Next iteration:
 - $2(v - 2)$ possibilities
 - Each vertex in T shared $v-1$ edges with other vertices, but the edges they shared with each other already in T
 - Next:
 - $3(v - 3)$ possibilities
 - ...
- Runtime:
 - $\sum_{i=1}^{v-1} (i * (v - i)) = \Theta(\text{largest term} * \text{number of terms})$
 - number of terms = v
 - largest term is $v^2/4$ (when $i=v/2$)
 - Evaluates to $\Theta(v^3)$

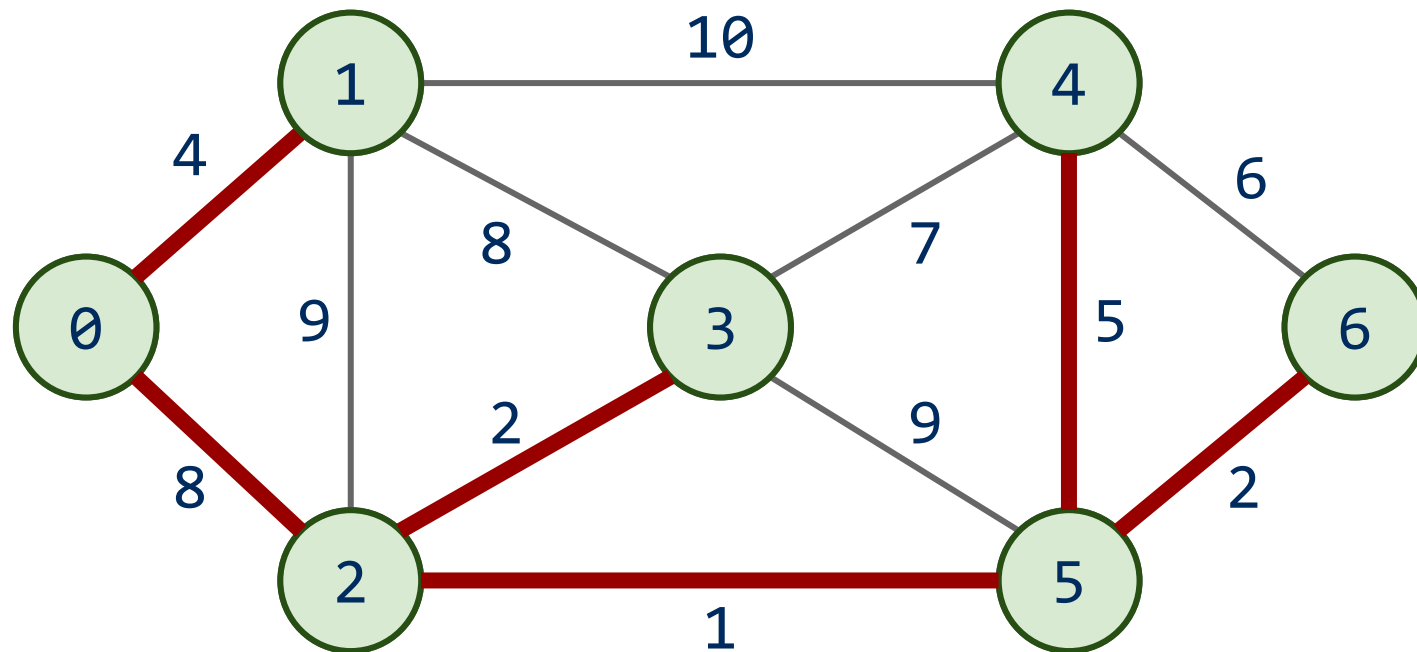
Do we need to look through all remaining edges?

- No! We only need to consider the *best* edge possible for each vertex!
 - The best edge of each vertex can be updated as we add each vertex to T

An enhanced implementation of Prim's Algorithm

- Add start vertex to T
- Search through the neighbors of the added vertex to adjust the parent and best edge arrays as needed
- Search through the best edge array to find the next addition to T
- Repeat until all vertices added to T

Prim's algorithm



	0	1	2	3	4	5	6
Parent:	--	0	0	2	5	2	5
Best Edge:	0	4	8	2	5	1	2

OK, so what's our runtime?

- For every vertex we add to T , we'll need to check all of its neighbors to update their best edges as needed
 - Let's assume we use an **adjacency matrix**:
 - Takes $\Theta(v)$ to check the neighbors of a given vertex
 - Time to update parent/best edge arrays?
 - $\Theta(1)$
 - Time to pick next vertex?
 - $\Theta(v)$
 - Total: $v \cdot \Theta(v) = \Theta(v^2)$

OK, so what's our runtime?

- For every vertex we add to T , we'll need to check all of its neighbors to update their best edges as needed
 - Let's assume we use **adjacency lists**
 - Takes $\Theta(d)$ to check the neighbors of a given vertex
 - Time to update parent/best edge arrays?
 - $\Theta(1)$
 - Time to pick next vertex?
 - $\Theta(v)$
 - Total: $v * \Theta(v + d) = \Theta(v^2)$

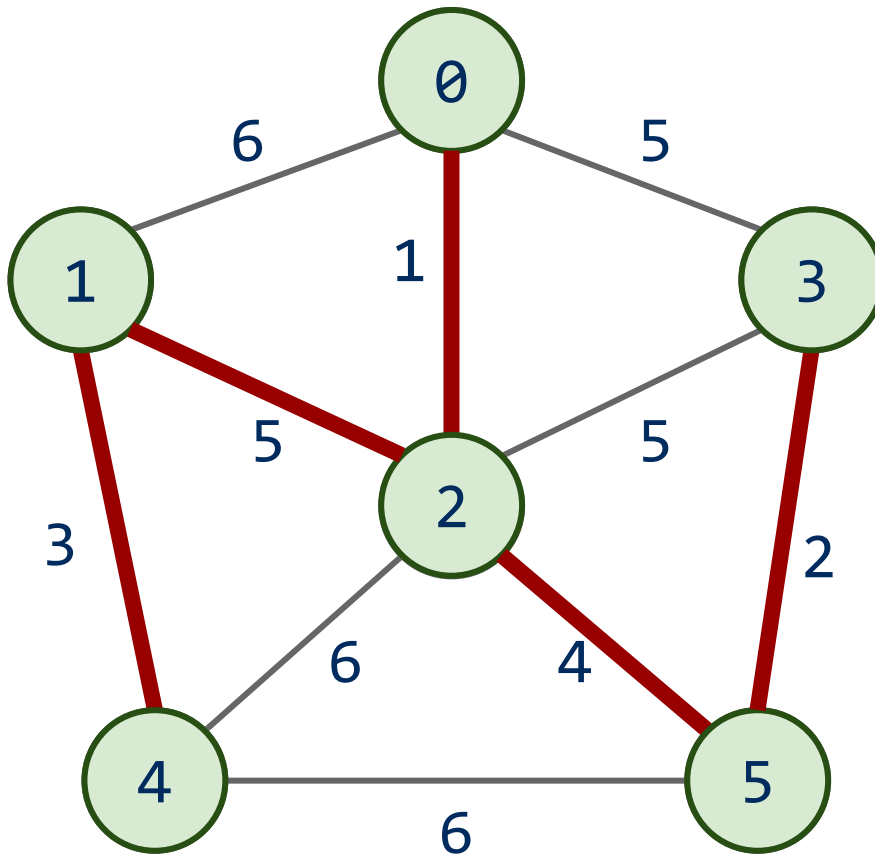
Prim's MST Algorithm

- seen, parent, and BestEdge are arrays of size v
 - Initialize seen to false, parent to -1, and BestEdge to infinity
 - BestEdge[start] = 0
 - for $i = 0$ to $v-1$
 - Find a vertex w with seen[w] = false and BestEdge[w] is the minimum over all unseen vertices
 - seen[w] = 1
 - for each neighbor x of w
 - if(BestEdge[x] > edge weight of edge (w, x)
 - BestEdge[x] = edge weight of (w, x)
 - parent[x] = w
- The parent array represents the found MST

What about a faster way to pick the best edge?

- Sounds like a job for a priority queue!
 - Priority queues can remove the min value stored in them in $\Theta(\lg n)$
 - Also $\Theta(\lg n)$ to add to the priority queue
- What does our algorithm look like now?
 - Visit a vertex
 - Add edges coming out of it to a PQ
 - While there are unvisited vertices, pop from the PQ for the next vertex to visit and repeat

Prim's with a priority queue



PQ:

1: (0, 2)

2: (5, 3)

3: (1, 4)

4: (2, 5)

5: (2, 3)

5: (0, 3)

5: (2, 1)

6: (0, 1)

6: (2, 4)

6: (5, 4)

Runtime using a priority queue

- Have to insert all e edges into the priority queue
 - In the worst case, we'll also have to remove all e edges
- So we have:
 - $e * \Theta(\lg e) + e * \Theta(\lg e)$
 - $= \Theta(2 * e \lg e)$
 - $= \Theta(e \lg e)$
- This algorithm is known as *lazy Prim's*

Do we really need to maintain e items in the PQ?

- I suppose we could not be so lazy
- Just like with the best edge array implementation, we only need the best edge for each vertex
 - PQ will need to be indexable to update the best edge
- This is the idea of *eager Prim's*
 - Runtime is $\Theta(e \lg v)$

Eager Prim's Runtime

$$\begin{array}{lcl} v & \text{insertions} & : v \log v \\ e & \text{updates} & : e \log v \\ v & \text{removals} & : v \log v \\ \hline & & (e+v) \log v = \Theta(e \log v) \end{array}$$

$e \geq (v-1)$

Comparison of Prim's implementations

- Parent/Best Edge array Prim's

- Runtime: $\Theta(v^2)$
- Space: $\Theta(v)$

- Lazy Prim's

- Runtime: $\Theta(e \lg e)$
- Space: $\Theta(e)$
- Requires a PQ

- Eager Prim's

- Runtime: $\Theta(e \lg v)$
- Space: $\Theta(v)$
- Requires an indexable PQ

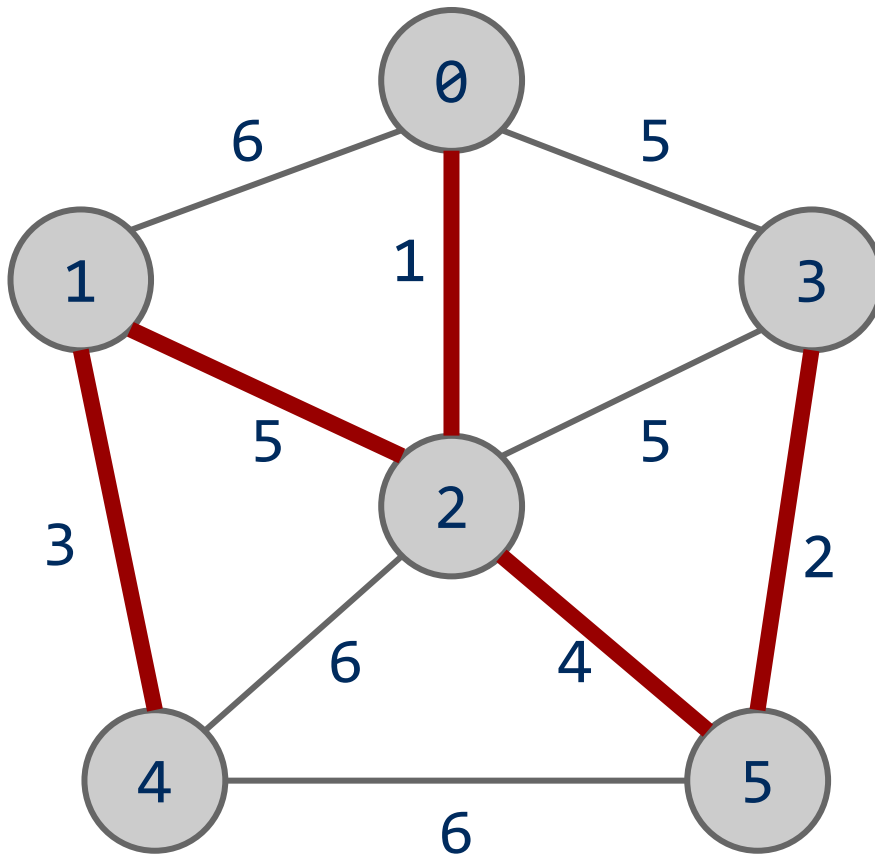
How do these compare?



Another MST algorithm

- Kruskal's MST:
 - Insert all edges into a PQ
 - Grab the min edge from the PQ that does not create a cycle in the MST
 - Remove it from the PQ and add it to the MST

Kruskal's example



PQ:

1: (0, 2)

2: (3, 5)

3: (1, 4)

4: (2, 5)

5: (2, 3)

5: (0, 3)

5: (1, 2)

6: (0, 1)

6: (2, 4)

6: (4, 5)

Kruskal's runtime

- Instead of building up the MST starting from a single vertex, we build it up using edges all over the graph
- How do we efficiently implement cycle detection?

Kruskal's Runtime

e iterations

remove

$\log e$

cycle
detection

$\Theta(v + e)$

DFS/BFS

$$e(v + e) = \Theta(e^2)$$